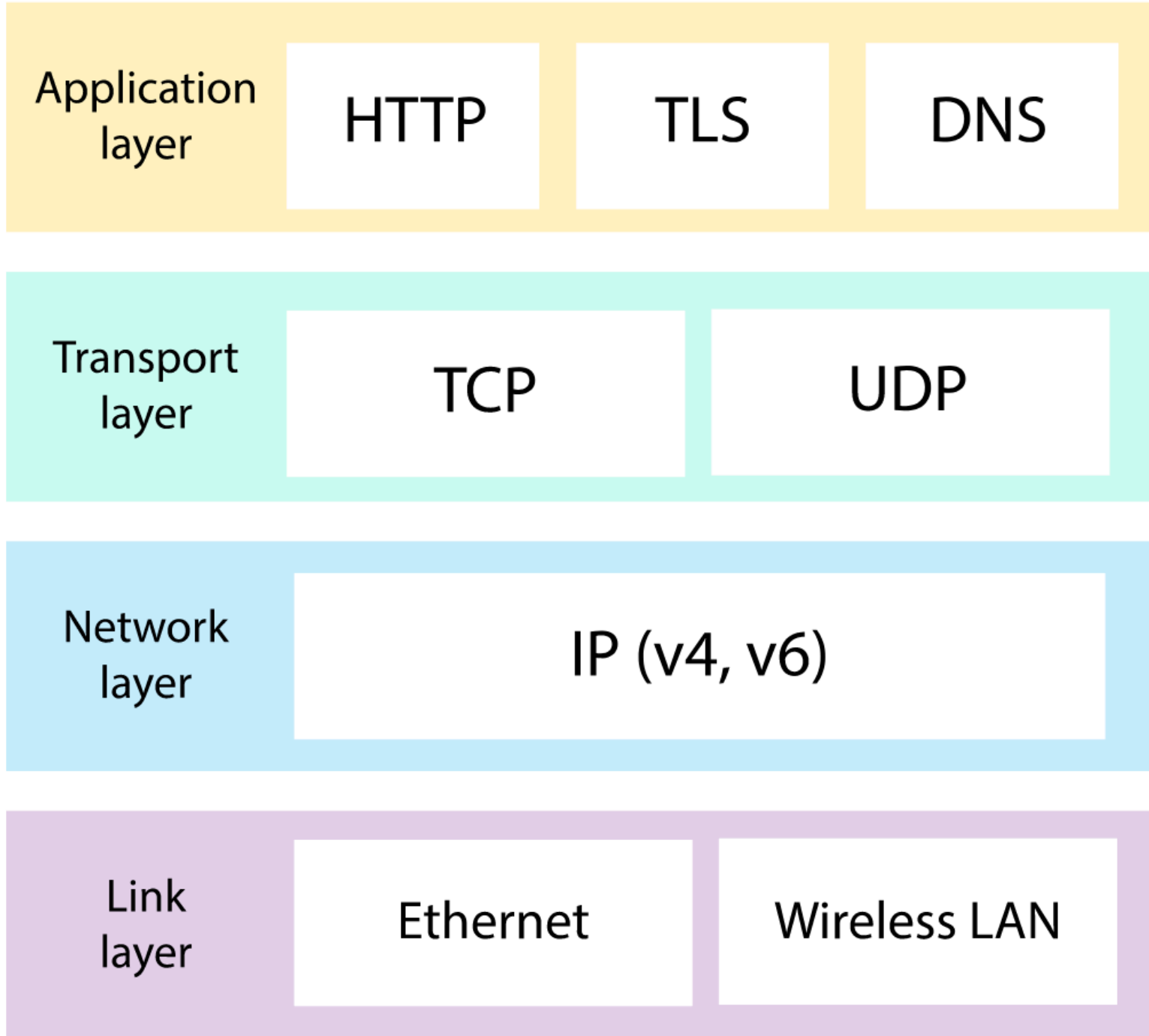
ITI0101 Sissejuhatus infotehnoloogiasse.

13. Web Applications II.

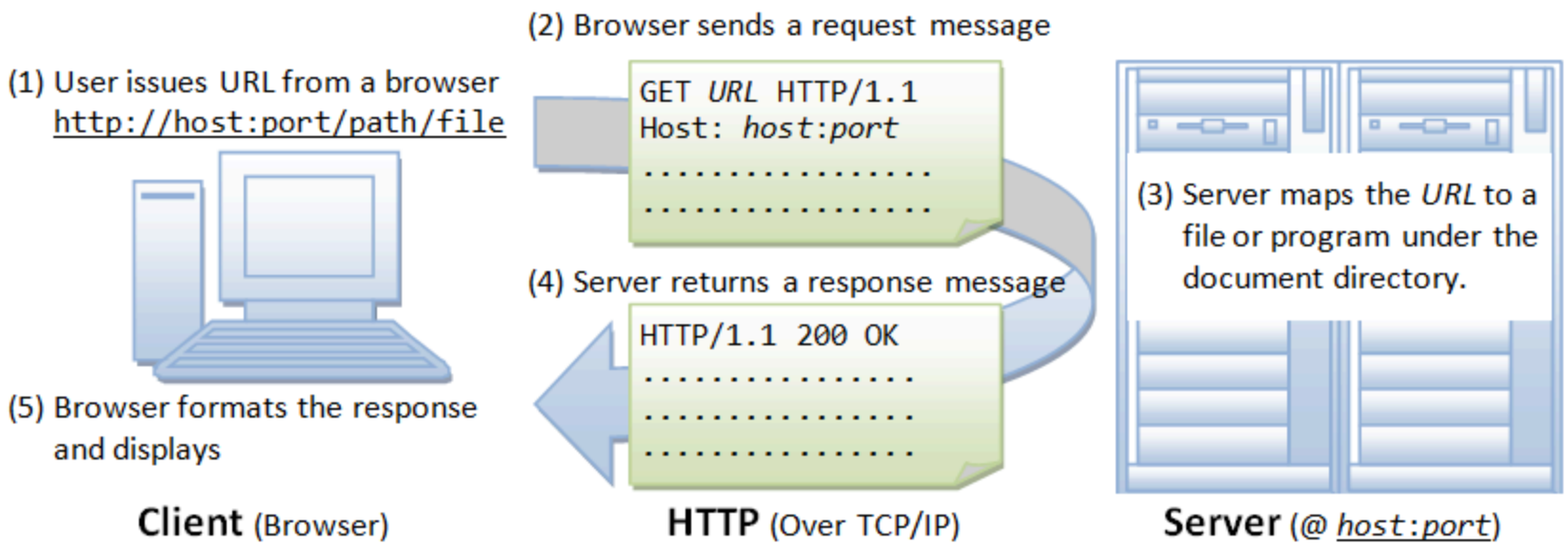
Martin Verrev
martin.verrev@taltech.ee


Application layer
HTTP
TLS
DNS
Transport layer
TCP
UDP
Network layer
IP (v4, v6)
Link layer
Ethernet
Wireless LAN

HTTP (Hypertext Transfer Protocol)

HTTP is perhaps the most popular application protocol used in the Internet.

- HTTP is an **asymmetric** request-response client-server protocol. An client sends a request message to a server; the server, in turn, returns a response message. The client pulls information from the server (instead of server pushing data to the client).
- HTTP is a **stateless** - the current request does not know what has been done in the previous requests.
- HTTP permits negotiating of data type and representation, so as to allow systems to be built independently of the data being transferred.


(1) User issues URL from a browser
http://host:port/path/file
(2) Browser sends a request message
GET URL HTTP/1.1
Host: host:port
.....
.....
(3) Server maps the URL to a file or program under the document directory.
(4) Server returns a response message
HTTP/1.1 200 OK
.....
.....
.....
(5) Browser formats the response and displays
Client (Browser)
HTTP (Over TCP/IP)
Server (@ host:port)

Uniform Resource Locator (URL)

Used to uniquely identify a resource over the web.

```
protocol://hostname:port/path-and-file-name
```

There are 4 parts in a URL:

- **Protocol:** The application-level protocol used by the client and server, e.g., HTTP, FTP, and telnet.
- **Hostname:** The DNS domain name or IP address of the server.
- **Port:** The TCP port number that the server is listening for incoming requests from the clients.
- **Path-and-file-name:** The name and location of the requested resource, under the server document base directory.

HTTP Request

A message sent by a client a server to request a resource. These requests follow a specific structure and contain essential components:

- HTTP Method: GET, POST, PUT, DELETE etc.
- Resouce URI
- Protocol
- Headers: Additional information about the request
- (optional) Body

```
GET /example-page HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
```

HTTP Response

Server’s reply to an HTTP request. HTTP responses also have a specific structure:

- Status Line
- Headers: Additional metadata about the response
- Body: The requested data or resource.

```
Header + Empty Line + Content
```

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Server: Apache
Content-Length: 3272
```

```
<!DOCTYPE html>
<html>
<!-- ... HTML content ... -->
</html>
```

Making Requests

Command Line: curl

```
curl https://news.ycombinator.com
curl https://news.ycombinator.com -i  > ajut.txt
```

Programmatically: (Python Example)

```
import requests
x = requests.get('https://news.ycombinator.com')
print(x.text)
```

```
import urllib.request
req = urllib.request.urlopen('http://news.ycombinator.com')
html = req.read()
print(html)
```

HTTP: GET

GET: Usually used for submitted search requests, or any request where you want the user to be able to pull up the exact page again.

Advantages of GET:

- URLs can be bookmarked safely.
- Pages can be reloaded safely.

Disadvantages of GET:

- Variables are passed through url as name-value pairs. (Security risk)
- Limited number of variables that can be passed.

HTTP: POST

POST: Used for higher security requests where data may be used to alter a database, or a page that you don't want someone to bookmark.

Advantages of POST:

- Name-value pairs are not displayed in url. (Security += 1)
- Unlimited number of name-value pairs can be passed via POST. Reference.

Disadvantages of POST:

- Page that used POST data cannot be bookmark.

Flask

Python micro framework for building web applications. It is designed to make getting started quick and easy, with the ability to scale up to complex applications.

https://flask.palletsprojects.com

Flask Server

Installation:

`sudo apt install python3 python3-flask` or `pip install flask`

Hello Flask: `hello.py`

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
        return "Hello World!"
```

Running Flask

Start Application:

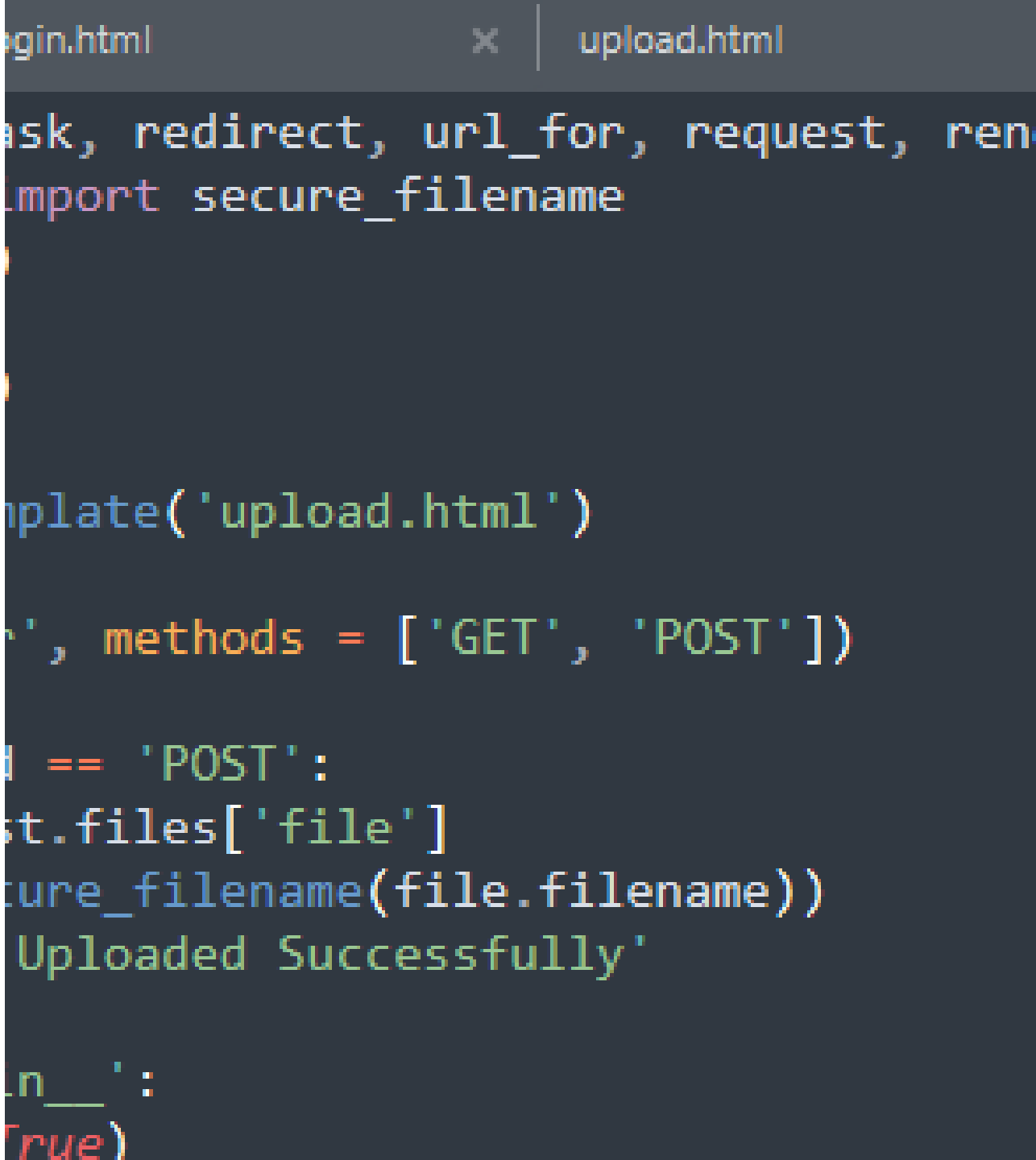
```
FLASK_APP=hello.py flask run
```

or

```
flask --app sum run
```

View in Browser

```
http://127.0.0.1:5000/
```



Serving Static Files

Teeme väikesed katsefailid `katse.txt` ja `katse.html`

Flask serveerib faile kataloogist static, teeme selle ja kopeerime:

```
mkdir static
cd static
cp /mnt/c/Users/Tanel/katse.txt .
cp /mnt/c/Users/Tanel/katse.html .
```

Vaata brauserist:

```
http://127.0.0.1:5000/static/katse.txt
http://127.0.0.1:5000/static/katse.html
```

data.py

Tee programm `data.py`:

```
from flask import Flask
from flask import request
app = Flask(__name__)
@app.route("/")
def hello():
  return "Hello World!"
@app.route('/data', methods=['POST', 'GET'])
def data():
  inparams={}
  keys=request.args.keys()
  for key in keys:
    inparams[key]=request.args.get(key)
  return(str(inparams))
```

URL: http://127.0.0.1:5000/data?a=1&b=2

sum.py

```
from flask import Flask
from flask import request
app = Flask(__name__)

@app.route('/sum', methods=['POST', 'GET'])
def sum():
  a = request.args.get("a",0)
  b = request.args.get("b",0)
  return(str(int(a)+int(b)))
```

URL: http://127.0.0.1:5000/sum?a=1&b=2

index.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
        <meta charset="UTF-8">
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <h1>Page Title</h1>

        Content

        <script src="script.js"></script>

    </body>
</html>
```

SPA

A single-page application (SPA) is a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server.

This approach avoids interruption of the user experience between successive pages, making the application behave more like a desktop application.

In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load,[1] or the appropriate resources are dynamically loaded and added to the page as necessary

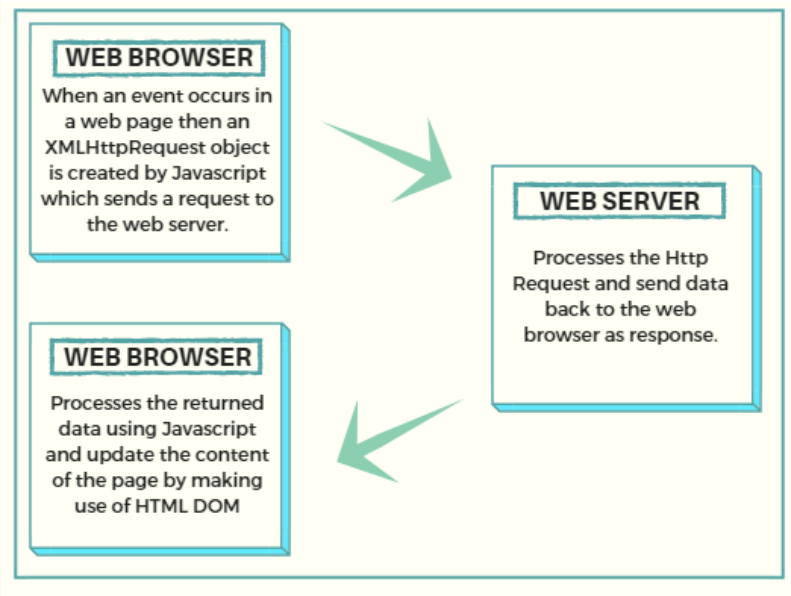
calc.html

```
<div id="ans"></div>
get sum:<p>
<form>
  a <input type='text' name="a" id='a'> <br>
  b <input type='text' name="b" id='b'> <p>
  <input type='button' onclick='calc()' value='calculate with javascript!'>
</form>
<script>
function calc() {
  var a,b,res;
  a=document.getElementById('a').value;
  b=document.getElementById('b').value;
  res=String(parseInt(a)+parseInt(b));
  document.getElementById('ans').innerHTML="Sum is: "+res;
}
</script>
```

AJAX

AJAX: Javascript + asynchronous queries.

Allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.



Javascript: fetch

`fetch()` allows you to make network requests similar to XMLHttpRequest (XHR). The main difference is that the Fetch API uses Promises, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest.

```
    fetch(url, {options})
        .then(response => response.json())
        .then(data => {
            //Do something
        })
        .catch(err => console.error(err));
```

ajax.html

```
<div id="ans"></div>
get sum:<p>
<form>
  a <input type='text' id='a'> <br>
  b <input type='text' id='b'> <p>
  <input type='button' onclick='calc()' value='calculate with ajax!'>
</form>
<script>
function calc() {
  var a,b,url;
  a=document.getElementById('a').value;
  b=document.getElementById('b').value;
  url="/sum?a="+a+"&b="+b
  fetch(url, {
    method: "get"
  }).then(r=>r.text()).then(handleresult);
}
function handleresult(r) {
  document.getElementById('ans').innerHTML="Sum is: "+r;
}
</script>
```

sum.py: Handle POST requests

sum.py

```
from flask import Flask
from flask import request
app = Flask(__name__)

@app.route('/sum', methods=['POST', 'GET'])
def sum():
  if request.method == 'POST':
      a = request.form.get("a",0)
      b = request.form.get("b",0)
  else:
    a = request.args.get("a",0)
    b = request.args.get("b",0)
  return(str(int(a)+int(b)))
```

URL : http://127.0.0.1:5000/sum?a=1&b=2

POST: Updated `data.py`

```
from Flask import jsonify

@app.route('/data', methods=['POST', 'GET'])
def data():
        inparams={}
        if request.method == 'POST':
                keys=request.form.keys()
                for key in keys:
                        inparams[key]=request.form.get(key)
        else:
                keys=request.args.keys()
                for key in keys:
                        inparams[key]=request.args.get(key)
        return jsonify(inparams)
```

Making POST requests programmatically

curl

```
curl -X POST http://127.0.0.1:5000/data
   -H "Content-Type: application/x-www-form-urlencoded"
   -d "param1=value1&param2=value2"
```

Python

```
import requests
url = 'http://127.0.0.1:5000/data'
myobj = {'somekey': 'somevalue'}
x = requests.post(url, json = myobj)
print(x.text)
```

Making POST requests programmatically

Using Fetch

```
fetch("http://127.0.0.1:5000/data", {
    method: "POST",
    body: JSON.stringify({
        title: "foo",
        body: "bar",
        userId: 1
    }),
    headers: {
        "Content-type": "application/json; charset=UTF-8"
    }
})
.then(response => response.json())
.then(json => console.log(json))
.catch(err => console.error(err));
```

Server Micro Framework Recommendations

Python:

- Flask: https://flask.palletsprojects.com/ or Bottle: https://bottlepy.org/

Node:

- Express: https://expressjs.com/

PHP:

- Slim: https://www.slimframework.com/ or Lumen: https://lumen.laravel.com/

Java:

- Spring Boot: https://spring.io/projects/spring-boot

Thank you!